LLNL-TR-637873

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

User-Specified and Automatic Data Layout Selection for Portable Performance

K. Sharma, I. Karlin, J. Keasler, J. McGraw, V. Sarkar

May 30, 2013

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

User-Specified and Automatic Data Layout Selection for Portable Performance

Technical Report LLNL-TR-637873, Lawrence Livermore National Laboratory, May 2013[*]

Kamal Sharma[§], Ian Karlin[†], Jeff Keasler[†], James R. McGraw[†], Vivek Sarkar[§]

[§]Rice University
Houston, Texas
United States
{kgs1,vsarkar}@rice.edu

[†]Lawrence Livermore National Laboratory
Livermore, CA 94551
United States
{karlin1,keasler1,mcgraw1}@llnl.gov

ABSTRACT

This paper describes a new approach to managing array data layouts to optimize performance for scientific codes. Prior research has shown that changing data layouts (e.g., interleaving arrays) can improve performance. However, there have been two major reasons why such optimizations are not widely used: (1) the need to select different layouts for different computing platforms, and (2) the cost of re-writing codes to use to new layouts. We describe a source-to-source translation process that allows us to generate codes with different array interleavings, based on a data layout specification. We used this process to generate 19 different data layouts for an ASC benchmark code (IRSmk) and 32 different data layouts for the DARPA UHPC challenge application (LULESH). Performance results for multicore versions of the benchmarks with different layouts show significant benefits on four computing platforms (IBM POWER7, AMD APU, Intel Sandybridge, IBM BG/Q). For IRSmk, our results show performance improvements ranging from 22.23× on IBM POWER7 to 1.10× on Intel Sandybridge. For LULESH, we see improvements ranging from 1.82× on IBM POWER7 to 1.02× on Intel Sandybridge. We also developed a new optimization algorithm to recommend a layout for an input source program and specific target machine characteristics. Our results show that the performance of this automated layout algorithm outperforms the manual layouts in one case and performs within 10% of the best architecture-specific layout in all the other cases, but one.

1. INTRODUCTION

This paper describes a framework and methodology to automatically improve the performance of scientific codes running on a variety of HPC computing platforms. "Portable performance" remains one of the most challenging problems for scientific application developers. Achieving good performance on a specific HPC platform often requires coding adjustments to fit a specific set of machine parameters e.g., cache size, cache line size, number of registers, main memory latency, memory bandwidth, etc. Unfortunately, adjustments for one platform often impedes performance on other platforms. This paper focuses on *data layout optimization*, which is steadily contributing a larger impact on performance. Most programming languages require developers to make array-of-struct (AoS) or struct-of-array (SoA) decisions (or combinations thereof) early in development. For long-lived applications, the following challenge can be encountered repeatedly (and now with increasing frequency): what to do when a new architecture is introduced with a memory subsystem that would benefit from a different data structure layout in the program? With current languages, a near-complete rewrite of an application at a low level is usually necessary, since each data access needs to be rewritten. Historically, developers of large scientific codes avoid changing data layouts because it involves changing too many lines of code, the expected benefit of a specific change is difficult to predict, and whatever works well on one system may hurt on another. Our approach demonstrates how these obstacles can be avoided.

The remainder of this paper describes the development of our scheme and the results obtained thus far. Section 2 describes a motivating example (IRSmk) with a single triple-nested loop and 29 array variables, and shows that changing array layouts can significantly impact performance on four different compute platforms. Section 3 introduces TALC, a source-to-source transformation tool and accompanying memory management runtime, that enables us to generate different data layouts based on guidance provided in a "meta file"', while imposing some restrictions on data accesses to ensure the legality of changing data layouts. Section 4 gives an overview of the LULESH mini-application. Section 5 presents a summary of empirical results obtained on four different platforms: IBM POWER7, AMD APU, Intel Sandybridge, and IBM BG/Q. We used the TALC tool to generate 19 different layouts for IRSmk, and 32 different layouts for LULESH. For IRSmk, our results show performance improvements ranging from 22.23× on IBM POWER7 to 1.10× on Intel Sandybridge. For LULESH, we see improvements ranging from 1.82× on IBM POWER7 to 1.02× on Intel Sandybridge. The insights gained from Section 5 led to the development of a new optimization algorithm (described in Section 6) to recommend a good layout for a given source program and specific target machine characteristics. Section 7 presents results from the automated layout algorithm showing that in all but one case it is within 10% of the best manual architecture-specific layout and in one case better. Finally, Section 8 summarizes related work, and Section 9

[*]Also available as Technical Report TR13-03, Department of Computer Science, Rice University, April 2013

contains our conclusions and plans for future work.

2. MOTIVATING EXAMPLE

We use the IRSmk benchmark (a 27-point stencil loop kernel in the ASC Sequoia Benchmark Codes [2]) as a motivating example to illustrate the impact of data layouts on performance. Figure 1 shows the main loop kernel of IRSmk. For simplicity, we ignore accesses to all arrays starting with the letter x, since they all alias to the same array with different offsets. We also ignore array b since it only occurs in a single write access. This leaves 27 static read accesses.

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
  for ( jj = jmin ; jj < jmax ; jj++ ) {
    for ( ii = imin ; ii < imax ; ii++ ) {
      i    = ii + jj * jp + kk * kp ;
      b[i] =  dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i]
            + dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i] * xdcr[i]
            + dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i]
            + cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i]
            + ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i]
            + cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i]
            + ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i]
            + ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i]
            + ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i];
} } }
```

Figure 1: IRSmk Source Code

As a preview of results to come, in Section 5 where we will look at more layouts and thread counts we look at four different array layouts here to illustrate the potential for performance gains on different platforms. The purpose is to reaffirm what has been observed in past work: changing data layouts can significantly impact execution time and its effects are platform specific.

The default layout is the one observed in Figure 1, where the 27 arrays are stored separately (27 × 1). A simple rewrite can change the layout by interleaving groups of three arrays, thus producing 9 actual arrays (9 × 3). Another rewrite can interleave 9 arrays each, producing three arrays (3 × 9). The final rewrite interleaves all 27 arrays into one array (1 × 27). We ran these four versions of IRSmk on four different platforms: IBM POWER7, AMD APU, Intel Sandybridge, and the IBM BG/Q, using a problem size of 100^3 and all cores on each platform. The results are presented in Table 1. All examples show positive gains for all of the layout options. However, the performance improvement varies dramatically across different platforms.

Platform	27 × 1	9 × 3	3 × 9	1 × 27
IBM POWER7	1.00	4.66	4.66	4.71
AMD APU	1.00	1.26	1.38	1.40
Intel Sandybridge	1.00	1.06	1.10	1.10
IBM BG/Q	1.00	1.65	2.14	2.20

Table 1: Performance improvement of different layouts relative to baseline 27 × 1 layout, for different platforms

3. TALC DATA LAYOUT FRAMEWORK

This section describes our extensions to the TALC Framework [4, 17] to support user-specified and automatic data layouts, driven by a Meta file specification. TALC stands for Topologically-Aware Layout in C. TALC is a source-to-source compiler translation tool and accompanying runtime system that dramatically reduces the effort needed to experiment with different data layouts. Our extended version of TALC has been implemented in the latest version of the ROSE [3] compiler infrastructure. In the process of extending TALC, we have re-implemented the entire source base, added new functionality of Automated Layouts and extended layout transformations.

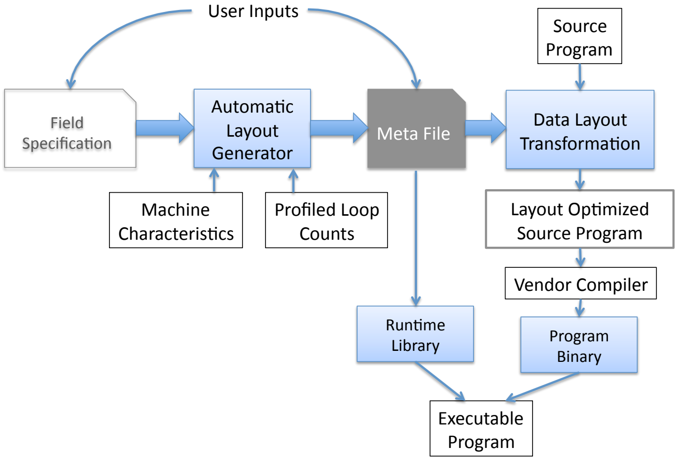


Figure 2: Extended TALC Framework

Figure 2 shows the overall framework. TALC can be configured to run in two modes: Automated Layout and Manual Layout. For both of these modes, a user needs to provide some input to perform data layout transformation. In the Automated Layout mode, the user provides a *field specification*. A field specification file is a simple schema file, which specifies arrays that should be considered for transformation. The field specification file is necessary because it enables our tool to only transform the specified arrays (like the 27 arrays in the IRSmk example discussed in Section 2). Figure 3 shows a sample field specification file. The View keyword is used internally to parse the data layouts. The field keyword specifies arrays considered for layout transformation. Each field has a type associated with it, specified by the : separator. In this example, d stands for the double data type. Specifying the data type helps with type checking array subscripts during layout transformations. More information on the Automatic Data Layout Selection will be provided in Section 6. For now, we will focus on the manual layout scheme.

The Meta file specifies the data layouts TALC should produce. A Meta file can be generated either automatically or manually. As an example, Figure 4 contains an example of a Meta file that can be used to drive user-specified data layouts. Unlike the field specification file in Figure 3., the Meta file also specifies which fields should be combined into the same array. So, this schema specifies that four arrays of structs are desired. For example, arrays x, y and z will be interleaved in a single array.

Data Layout transformation is a key component in the TALC framework. The transformation accepts a C/C++ source program and Meta file, produces an equivalent program and changes the data layout of the specified arrays to match the

```
View node
{
   Field {x:d}
   Field {y:d}
   Field {z:d}
...
}
```

Figure 3: Sample Field Specification file

```
View node
{
   Field {   x:d,   y:d,   z:d }
   Field {  xd:d,  yd:d,  zd:d }
   Field { xdd:d, ydd:d, zdd:d }
   Field {  fx:d,  fy:d,  fz:d }
}
```

Figure 4: Sample TALC Meta file

Meta file. The layout transformation matches the names and data type of the arrays before modifying the source code. Array subscripts are automatically handled. The layout transformation also rewrites the memory allocation of the layout transforming arrays to a library call. This call is made at the runtime thereby handling memory allocation gracefully for the entire group in a field. The runtime library ensures memory-aligned allocation for the array grouping. Figure 5 shows the key portion of an input file. Figure 6 shows a stylized output file (the new array names are not a part of TALC) generated by the layout transformation, based on the Meta file in Figure 4.

To ensure that data layout transformations can safely be performed TALC is that it imposes some programming restrictions in C/C++:

- All accesses to candidate arrays for data layout transformation must be written as *array*[*index*]. The alternate form, *index*[*array*] is prohibited.
- All “aliases” for the same array must use the same name. This is especially important when passing arrays by reference across functions.
- All arrays in the same field group (as specified in the Meta file) must be of the same length.
- Only single dimensional arrays are currently supported. Transformation of multi-dimensional arrays is a topic for future work.

4. LULESH MINI-APPLICATION

Our application case study of data layout optimizations focuses on the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-application [5]. Hydrodynamics is widely used to model continuum material properties and material interactions in the presence of applied forces. The hydrodynamics portion of an explicit time-stepping multi-physics application might typically consume one third of application runtime. The LULESH mini-app provides a significantly simplified source code ($\sim$ 2600 lines) that illustrates the primary arrays needed for the computation and the structure of the loops that define how all of the arrays are accessed to complete the calculation. The version of the code used in these experiments uses OpenMP to run on multiple cores of a single node computing device.

```
for (int Node=0; Node<numNode; ++Node) {

// Calculate new Accelerations for the Node
   xdd[Node] = fx[Node] / Mass[Node];
   ydd[Node] = fy[Node] / Mass[Node];
   zdd[Node] = fz[Node] / Mass[Node];

// Calculate new Velocity for the Node
   xd[Node] += xdd[Node] * dt ;
   yd[Node] += ydd[Node] * dt ;
   zd[Node] += zdd[Node] * dt ;

//  Calculate new Position for the Node
   x[Node] += xd[Node] * dt ;
   y[Node] += yd[Node] * dt ;
   z[Node] += zd[Node] * dt ;
}
```

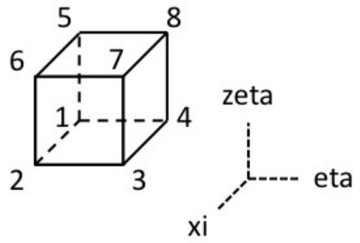
Figure 5: Sample C input file.

```
for (int Node=0; Node<numNode; ++Node) {

// Calculate new Accelerations for the Node
   Acc[Node].xdd = force[Node].x / Mass[Node];
   Acc[Node].ydd = force[Node].y / Mass[Node];
   Acc[Node].zdd = force[Node].z / Mass[Node];

// Calculate new Velocity for the Node
   Vel[Node].xd += Acc[Node].xdd * dt ;
   Vel[Node].yd += Acc[Node].ydd * dt ;
   Vel[Node].zd += Acc[Node].zdd * dt ;

//  Calculate new Position for the Node
   Pos[Node].x += Vel[Node].x * dt ;
   Pos[Node].y += Vel[Node].y * dt ;
   Pos[Node].z += Vel[Node].z * dt ;
}
```

Figure 6: Stylized TALC output file.

LULESH arrays represent the 3D hexahedral mesh structure that is simulating 3D physical space. See Figure 7. Three types of data are represented in LULESH: element-centered, node-centered and symmetry. The 19 element-centered arrays represent information about the space modeled by a hexahedron (e.g., pressure, energy, volume, and mass). The 13 node-centered arrays represent information about the corners of each hexahedron (e.g, position: x, y, z, and velocity: xd, yd, zd). The 3 symmetry arrays represents information about all six outside surfaces of the physics problem (not the surfaces of each element). The problem size is set by a single parameter: the number of elements in one dimension of the physical space. The problem is set up as a cube, so a problem size of 45 means 45^3 total elements, 46^3 total nodes ,and 46^2 symmetry locations.

The “unstructured” characteristic of LULESH imposes constraints on the representation of node, element and symmetry data. See Figure 8 for a 3D example of an unstructured mesh. At various points on the surface, three, four or five elements may share a node. As a result, each element and each node are assigned a unique number. Element data is organized as a linear array, indexed by the element number. Nodal data is handled in an identical fashion. LULESH also contains a number of indirection arrays that enable looking up the eight nodes that comprise the corners of a given element and looking up the (up to) six elements sharing a face with a given element. One additional historical factor influenced the exact data layout in LULESH. LULESH attempts to mimic how production hydro codes are written. Such



Face	Nodes
1	1 2 3 4
2	5 8 7 6
3	4 8 5 1
4	2 6 7 3
5	1 5 6 2
6	3 7 8 4

Figure 7: LULESH grid example, one element with its eight nodes.

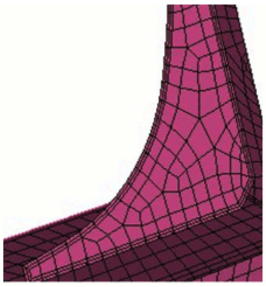


Figure 8: Example of an unstructured mesh. An unbounded number of elements can share a node.

codes are often optimized for vector-parallel execution. To maximize vectorization, many arrays that would normally be considered 3D quantities (e.g., position, velocity, acceleration, force) are stored as separate arrays (x, y, z, xd, yd, zd, ...).

The LULESH algorithm provides a significant challenge for data layout optimizations due to the physics-based uses of the various arrays within the loops of the program. The version of LULESH used in this study has undergone a variety of optimizations from the original published code, including aggressive loop fusion and elimination of temporary arrays [16]. The main loop is a sequential time-step loop that has three parts. Part one calculates new values for all node-centered variables ($\sim$ 60% of compute time). Part two computes updated values for all element-centered data($\sim$ 40% of compute time). Part three is a global reduce to determine how large of a time step can be taken for the next iteration($<$ 1% of compute time). Inside the time-step loop, the algorithm is almost fully parallel with 12 OpenMP parallel loops.

5. MANUAL LAYOUT RESULTS

To show the impact of data layouts on performance we ran experiments using our two test codes on four different platforms. The Linux-based systems are: *Power7*, *AMD APU*, *Intel Sandybridge* and *IBM BG/Q*. Table 2 summarizes the specifications of these architectures and compiler options used. For the *AMD APU*, we focused on the CPU and ignored the GPU. The two codes (IRSmk and LULESH) were both run in double precision on varying thread counts on both platforms. Specifically, we ran IRSmk on a problem based on a 100^3 mesh for 500 iterations. LULESH was run with a problem size = 90 (i.e. 90^3 elements and 91^3 nodes). Both of these benchmarks use OpenMP for parallelism, but do not allocate data in a Non Uniform Memory Access (NUMA) aware manner and therefore we limit our analysis to a single socket in the multi-socket Power 7 and Sandybridge architectures.

For both codes, TALC enabled testing a vast number of layouts, 19 for IRSmk and 31 for LULESH. To perform the layout transformations in IRSmk, between 56 and 272 (82%) lines of the original 330 lines of code were changed. For the LULESH the numbers are 98 to 477 (18%) lines of the original 2640. By using TALC, we not only were able to reduce the arduous effort of performing manual changes, but also eliminate the possibility of subtle bugs.

For each benchmark, we conducted extensive experiments across different layouts on four architectures. Figure 9 shows the nine IRSmk layouts for which we present results. Each row ”block” represents a set of arrays(identified by column) that have been interleaved. So, for example layout 10 is nine real arrays containing three interleaved arrays each. The names in each block are added simply for clarity of this presentation. Tables 3, 4, 5 and 6 shows the results for running IRSmk using selected thread counts on all nine layouts on each of the four platforms. For each test case, we report the speedup (or slowdown) of each layout against the “base case” which is the original code, running with an equivalent number of threads.

To understand how layouts impact the performance of IRSmk one first needs to understand that this kernel should be memory bound. Each iteration of the innermost loop in Figure 1 reads one unique double from 27 arrays, writes a single value to the b array and may read data part of x from memory if it is not in cache already. In addition, to these data reads only 53 FLOPs are performed along with 4 integer operations. Since each array is about 9MB, other than x there is no chance of any other array staying within cache between iterations. Therefore, performance should be limited by memory bandwidth. However, we see that except for Sandybridge significant speedups occur at all thread counts due to data layouts.

To better understand what is happening we looked at the total bandwidth requirement for moving all 29 arrays either from or to memory. For the 500 iterations this is about 119 GB of data motion. For now we ignore that x might be moved multiple times from memory and assume good caching and determine the bandwidth limited runtime of each system using the Stream Triad numbers [19]. For Sandybridge which has about 40 GB/s stream bandwidth this is just under 3s. For BG/Q with about 28 GB/s of bandwidth this is 4.25s. For the AMD APU with about 15 GB/s this is just under 8 s. For the Power 7 there is about 13 GB/s of bandwidth implying a lower bound of 9.15s. Therefore, we have a best case runtime for each machine.

The results of the best layout for IRS on all machines show performance of at least 70% of optimal and over 95% on Sandybridge. For Sandybridge the best layout is 3.05s, for the AMD APU it is 10.04 s, for BG/Q it is 5.2 s and for the Power 7 it is 12.52 s. BG/Q might perform slightly worse due to in-order cores not hiding as much latency as the other

Table 2: Architecture and compiler specifications

Machine	Architecture Specification	Compiler Option
IBM Power7	Quad IBM Power 7 (eight-core 3.55 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache, 32MB L3 Cache)	`xlc v11.1 -O3 -qsmp=omp -qthreaded -qhot -qtune=pwr7 -qarch=pwr7`
AMD APU	Single AMD A10-5800K APU (quad-core 3.8 GHz processor, 16KB L1 D-Cache per core, 4MB L2 Cache)	`gcc v4.7.2 -O3 -fopenmp`
Intel Sandybridge	Dual Intel E5-2670 Sandybridge CPU (eight-core 2.6 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache per core, 20MB L3 Cache)	`icc v12.1.5 -O3 -fast -parallel -openmp`
IBM BG/Q	16 IBM PowerPC A2 cores/node, 1.6 GHz processor, 32 MB eDRAM L2 cache	`gcc v.4.4.6 -O3 -fopenmp`

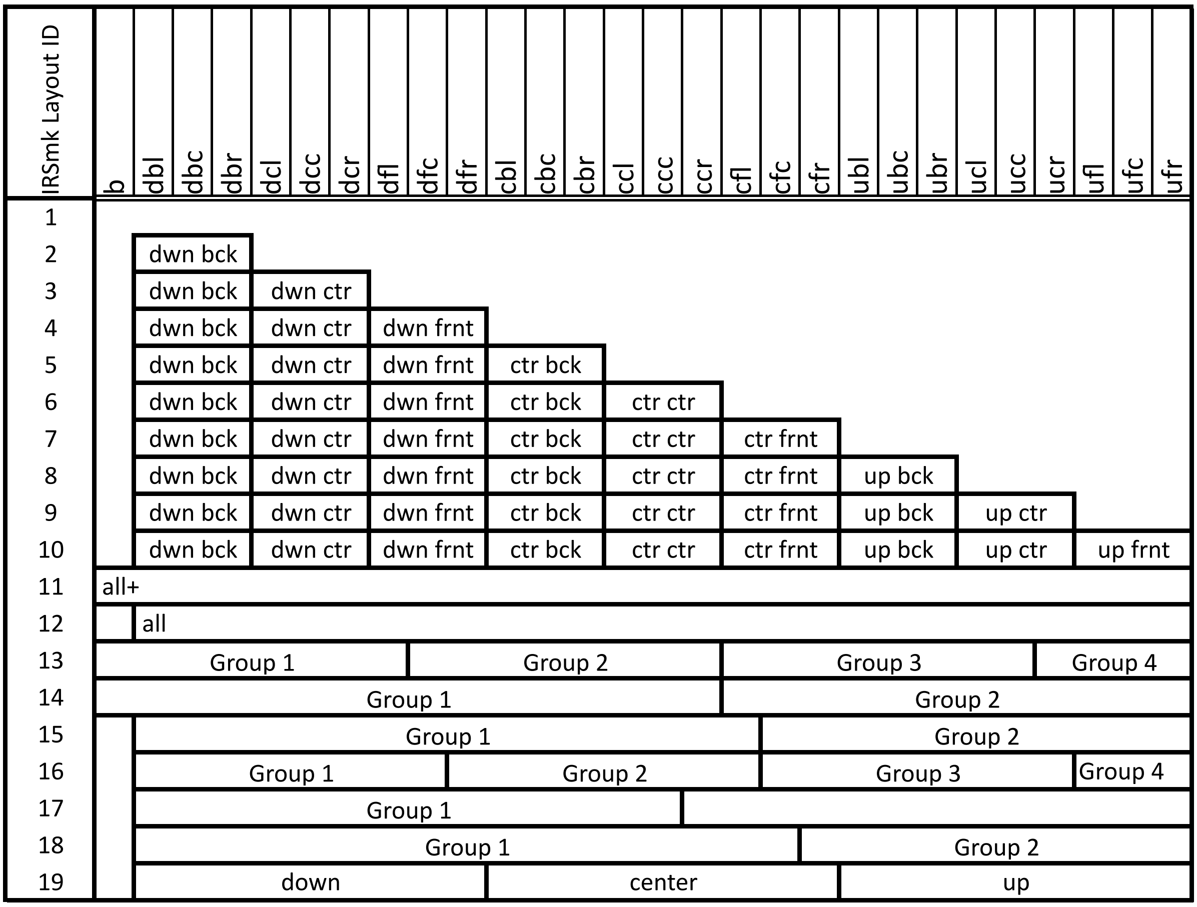

IRSmk Layout ID
b dbl dbc dbr dcl dcc dcr dfl dfc dfr cbl cbc cbr ccl ccc ccr cfl cfc cfr ubl ubc ubr ucl ucc ucr ufl ufc ufr
1
2 dwn bck
3 dwn bck dwn ctr
4 dwn bck dwn ctr dwn frnt
5 dwn bck dwn ctr dwn frnt ctr bck
6 dwn bck dwn ctr dwn frnt ctr bck ctr ctr
7 dwn bck dwn ctr dwn frnt ctr bck ctr ctr ctr frnt
8 dwn bck dwn ctr dwn frnt ctr bck ctr ctr ctr frnt up bck
9 dwn bck dwn ctr dwn frnt ctr bck ctr ctr ctr frnt up bck up ctr
10 dwn bck dwn ctr dwn frnt ctr bck ctr ctr ctr frnt up bck up ctr up frnt
11 all+
12 all
13 Group 1 Group 2 Group 3 Group 4
14 Group 1 Group 2
15 Group 1 Group 2
16 Group 1 Group 2 Group 3 Group 4
17 Group 1
18 Group 1 Group 2
19 down center up

Figure 9: IRSmk layouts selected for discussion.

Table 3: Speedup relative to base case on IBM Power 7 for IRSmk

IRSmk Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
1	1.02	1.01	1.02	0.99	0.94	1.12
2	1.16	1.14	1.17	1.13	1.08	1.28
3	1.33	1.32	1.34	1.30	1.24	1.47
4	1.59	1.57	1.60	1.55	1.53	1.78
5	2.46	2.41	2.45	2.33	2.35	2.68
6	2.86	2.82	2.84	2.66	2.65	3.01
7	3.01	2.96	2.99	2.78	2.75	3.11
8	5.43	5.26	5.16	4.10	4.11	4.38
9	9.82	9.14	7.94	4.66	4.57	4.71
10	14.66	13.50	9.24	4.66	4.08	4.62
11	10.00	9.40	7.80	4.37	3.84	3.95
12	9.88	9.45	8.45	4.71	4.91	5.02
13	13.47	11.91	8.57	4.59	4.02	4.30
14	15.24	12.54	8.27	4.38	3.71	3.77
15	19.14	16.20	9.32	4.67	4.47	4.93
16	14.30	12.73	8.84	4.67	4.42	4.78
17	18.83	15.80	9.32	4.67	4.49	4.93
18	13.90	12.65	9.06	4.70	4.67	4.92
19	22.23	17.18	9.33	4.66	4.27	4.78

Table 4: Speedup relative to base case on AMD APU for IRSmk

IRSmk Layout ID	1 Thread	2 Threads	4 Threads
1	1.00	1.00	1.02
2	1.07	1.03	1.07
3	1.14	1.06	1.12
4	1.21	1.11	1.16
5	1.26	1.16	1.19
6	1.38	1.24	1.22
7	1.45	1.30	1.23
8	1.40	1.33	1.25
9	1.33	1.34	1.25
10	1.21	1.26	1.26
11	1.63	1.45	1.17
12	1.38	1.50	1.40
13	2.26	1.55	1.12
14	1.99	1.57	1.14
15	1.74	1.66	1.47
16	2.30	1.76	1.43
17	1.58	1.68	1.46
18	1.49	1.55	1.39
19	1.62	1.60	1.38

Table 5: Speedup relative to base case on Intel Sandybridge for IRSmk

IRSmk Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1	1.00	1.01	1.01	1.01	1.01
2	1.07	1.08	1.07	1.02	1.01
3	1.17	1.18	1.16	1.03	1.01
4	1.24	1.26	1.22	1.03	1.02
5	1.31	1.31	1.26	1.05	1.01
6	1.35	1.36	1.28	1.05	1.02
7	1.39	1.40	1.30	1.05	1.02
8	1.42	1.43	1.32	1.06	1.00
9	1.35	1.37	1.32	1.07	1.03
10	1.20	1.22	1.25	1.07	1.03
11	1.19	1.16	1.00	0.86	0.84
12	1.28	1.26	1.19	1.10	1.06
13	1.54	1.47	1.05	0.85	0.80
14	1.43	1.29	0.90	0.76	0.74
15	1.48	1.46	1.30	1.11	1.05
16	1.62	1.60	1.36	1.10	1.03
17	1.47	1.45	1.30	1.11	1.05
18	1.46	1.44	1.29	1.11	1.05
19	1.63	1.60	1.35	1.10	1.05

Table 6: Speedup relative to base case on BG/Q for IRSmk

IRSmk Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.09	1.09	1.09	1.08	1.07	1.10	1.08
3	1.22	1.23	1.23	1.23	1.21	1.29	1.19
4	1.09	1.09	1.09	1.09	1.09	1.22	1.28
5	1.17	1.16	1.16	1.17	1.17	1.32	1.40
6	1.64	1.57	1.57	1.57	1.55	1.49	1.50
7	1.86	1.86	1.86	1.86	1.85	1.72	1.68
8	1.98	1.98	1.98	1.97	1.98	1.84	1.80
9	1.82	1.82	1.82	1.82	1.82	1.66	1.79
10	1.34	1.34	1.34	1.33	1.33	1.24	1.65
11	2.92	2.92	2.92	2.90	2.87	2.13	1.71
12	2.99	2.99	2.99	2.98	2.98	2.69	2.20
13	2.87	2.88	2.88	2.88	2.88	2.34	1.85
14	2.99	2.99	2.99	2.98	2.93	2.11	1.68
15	2.99	2.99	2.98	2.98	2.98	2.66	2.16
16	2.82	2.82	2.81	2.81	2.82	2.60	2.08
17	3.00	3.00	3.00	3.00	3.00	2.66	2.17
18	3.02	3.02	3.02	3.02	3.01	2.64	2.16
19	2.98	2.98	2.98	2.97	2.98	2.61	2.14

processors, while the AMD APU could be hurt by less data in the x array staying in its smaller cache. Finally, all the processors could be limited in their handling of the unequal amount of read and write data in IRSmk.

While the best case scenarios for each processor are similar, the base case is significantly different. On the Sandybridge chip, data layouts only sped up the computation by 1.11×. However, the initial code was running at over 85% of peak memory bandwidth, so a large speedup is not possible. On the other processors, performance is significantly worse when no data layouts are applied. From looking at the hardware specifications the largest difference in this regard is the number of hardware prefetch streams each processor can keep active at once. The SandyBridge can handle 32 per core [14], while BG/Q can handle 16 per core [10], the Power7 can handle 10 per core [22] and the AMD APU can handle 12 per core [6]. Therefore, the Sandybridge processor can handle all the arrays in the computation at once. However on other processors fusing arrays decreases the number of streams coming from memory, resulting in fewer data elements read in a latency-bound manner. This is especially important for in-order cores like BG/Q. Also the Power 7 benefits significantly from fewer arrays. Other layouts that are not shown show a trend of steadily improving performance as more arrays are grouped. Also when a calculation is latency bound, such as the base case, doubling the thread count halves runtime, by doubling the number of latency bound reads occurring concurrently. However, the memory bound layouts, that do not use all the prefetchers have their runtime barely decrease, such as layout 19, from 2 to 8 threads.

A related trend is that improvements from data layouts are more significant at lower core counts. This implies two conclusions. First compute-bound codes also benefit from data layout transformations. In the case of Sandybridge where there are enough stream prefetchers for the base code and enough bandwidth to feed a few, but not all cores merging arrays reduces the number of registers used as pointers by the compiler resulting in fewer instructions and possibly fewer pipeline stalls. Another benefit is that the number of elements accessed each loop from an array can be matched to cache boundaries, such as layout 16. The second observation is that for processors with an under provisioning of prefetchers when fewer cores are used the computation becomes latency-bound. With fewer cores to issue memory requests the memory bus becomes idle for a larger percentage of the time. Therefore, bandwidth is used less efficiently allowing for larger speedups when the core uses it more effectively.

A final observation is that not merging read only arrays in a loop with arrays that are written to increases the performance significantly. For IRSmk, when b was not merged with other groups performance was better in all cases except some single threaded examples as compared to combining it with other arrays. The performance difference between layouts 11 and 12 shows this effect. Modern architectures, such as AMD APU, often implement a write buffer to combine multiple writes to the same cache line to reduce the amount of data sent to main memory. This optimization is known as Write-Combining [1].

Tables 7, 8, 9 and 10 shows LULESH results for running the 31 layouts presented in Figure 10 on our four test platforms. For each test case, we report the speedup (or slowdown) of each layout against the "base case" which is the original code, running with an equivalent number of threads.

Data layout transformations on LULESH were less profitable overall than for IRSmk. This is not surprising since some arrays in LULESH are used together in certain places and not together in others. Therefore, combining them together will help and hurt performance simultaneously. For example, layout 24 combines all four triples of x, y , z values together. Many of these triples are used together in many functions, but not all. However, most of the time layout 27 that leaves the triples separate is faster. A notable exception is the Power7 for a single thread, which has the most cache, but the least bandwidth. It also suffers the most from not getting good prefetching as shown by the IRSmk results. Probably due to a combination of these reason combining these arrays that are often accessed indirectly, helps result in fewer latency bound reads and fewer streams of data being accessed helps on this chip.

The most interesting result from LULESH is that in most cases it seems the code not the hardware is dictating the best data layout. On the AMD APU, Intel Sandybridge and BG/Q the list of the best layouts always includes 28, 29 and 31. However, the Power7 is an outlier with its best layout varying across thread counts by a significant margin for the reasons explained above.

For LULESH as with IRSmk data layouts impacted the Sandybridge system the least with the largest speedup seen being only 1.02×. There are a few likely reasons for this. First as with IRSmk the Sandybridge architecture should be able to prefetch many streams at once. Also, in the case of bundling indirect accesses the large re-order window of the Sandybridge might hide memory latency better than the other chips. Finally, the Intel compiler used on this platform was the best at generating SIMD instructions for some of the compute bound loops of LULESH. Some of the data transformations result in the compiler no longer generating SIMD instructions and, therefore, while data layouts save on data motion in memory-bound portions of the code they can hurt performance in the compute bound sections.

6. AUTOMATIC DATA LAYOUT SELECTION

In this section, we describe the automatic data layout selection algorithm. The algorithm takes in a user-written field specification file and uses a greedy approach to automatically construct a data layout based on the architecture and input program. In this section we describe the machinery and equations used by the algorithm and then the algorithm itself.

6.1 Use Graph

The automated analysis begins by creating a mapping of all arrays used within each loop of the source program. We are only interested in determining which arrays appear inside each loop, not the exact location or order of use. So, our Use graph is a mapping from each array name to a function name along with the loop for the reference. In the case of nested loops, each array points to the inner-most loop in

	Nodal Arrays													Symmetry			Element Arrays																		
Layout ID	x	y	z	xd	yd	zd	xdd	ydd	zdd	fx	fy	fz	nodalMass	symmX	symmY	symmZ	lxim	lxip	letam	letap	lzetam	lzetap	ql	qq	p	q	elemBC	e	ss	elemMass	delv	volo	v	vdov	arealg
1																																			
2	Pos																																		
3	Pos			Vel																															
4	Pos			Vel			Acc																												
5	Pos			Vel			Acc			Force																									
6	Pos			Vel			Acc			Force				Symm																					
7	Pos			Vel			Acc			Force				Symm											p_q										
8	Pos			Vel			Acc			Force				Symm											p_q							v_vo			
9	Pos			Vel			Acc			Force				Symm											p_q				ss_em			v_vo			
10	Pos			Vel			Acc			Force				Symm											p_q				ss_em			v_vo		v_a	
11	PV						Acc			Force				Symm											p_q				ss_em			v_vo		v_a	
12	PVA									Force				Symm											p_q				ss_em			v_vo		v_a	
13	PVAF													Symm											p_q				ss_em			v_vo		v_a	
14	PVAF													Symm									ql_qq		p_q				ss_em			v_vo		v_a	
15	PVAF													Symm			l_pm						ql_qq		p_q				ss_em			v_vo		v_a	
16	PVAF													Symm			lee						ql_qq		p_q		lee		ss_em			v_vo		v_a	
17	PVAF													Symm			leeq								p_q		leeq		ss_em			v_vo		v_a	
18	PVAF													Symm			pqlq												ss_em			v_vo		v_a	
19	PVAF													Symm			Group1												ss_em			Group1		v_a	
20	PV						AF							Symm			pqlq												ss_em			v_vo		v_a	
21	PV						AF							Symm			lee						Group1				lee		ss_em			Group1		v_a	
22	PV						Group1							Symm			lee						Group2				lee		ss_em			Group2		v_a	
23	PV						Group1							Symm			lee						Group2				lee		Group3			Group2		v_a	
24	Group1													Symm			Group2																		
25	Pos			Vel			Acc			Force				Symm			l_pm																		
26	Group1													Symm			l_pm						Group2												
27	Pos			Vel			Acc			Force				Symm			Group1																		
28	Pos			Vel			Acc			Force				Symm			l_pm						ql_qq		p_q				ss_em			v_vo		v_a	
29	Pos			Vel			Acc			Force				Symm			l_pm						ql_qq		p_q		Group1					v_vo		v_a	
30	Pos			Vel			Acc			Force				Symm			l_pm						ql_qq		p_q										
31	Pos			Vel			Acc			Force				Symm			l_pm						ql_qq		p_q		eBC		ss_em			v_vo		v_a	
Auto	Pos			Vel			Group1							Symm			Group2												Group3						

Figure 10: LULESH layouts selected for discussion.

Table 7: Speedup relative to base case on IBM Power 7 for LULESH

LULESH Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
1	0.86	0.88	0.85	0.84	0.96	0.99
2	1.00	1.07	1.06	1.03	1.12	1.14
3	1.14	1.13	1.05	1.08	1.17	1.18
4	1.14	1.07	1.12	1.08	1.16	1.17
5	1.17	1.10	1.08	1.04	1.19	1.17
6	1.10	1.16	1.08	1.08	1.23	1.17
7	1.16	1.15	1.13	1.14	1.24	1.19
8	1.18	1.26	1.24	1.10	1.27	1.26
9	1.26	1.34	1.32	1.16	1.34	1.31
10	1.29	1.28	1.25	1.20	1.45	1.36
11	1.34	1.43	1.30	1.31	1.41	1.36
12	1.33	1.41	1.36	1.16	1.33	1.25
13	1.28	1.27	1.20	1.02	1.16	1.00
14	1.33	1.30	1.24	1.05	1.19	1.03
15	1.47	1.45	1.25	1.07	1.22	1.05
16	1.46	1.58	1.46	1.14	1.29	1.07
17	1.61	1.44	1.47	1.12	1.29	1.06
18	1.47	1.44	1.36	1.09	1.25	1.03
19	1.45	1.56	1.39	1.00	1.11	0.98
20	1.53	1.49	1.46	1.27	1.45	1.23
21	1.69	1.50	1.44	1.26	1.48	1.25
22	1.68	1.51	1.42	1.23	1.42	1.22
23	1.51	1.48	1.37	1.14	1.39	1.15
24	1.54	1.36	1.19	0.84	0.92	0.84
25	1.11	1.12	1.08	1.11	1.26	1.19
26	1.44	1.39	1.30	0.90	1.02	0.90
27	1.44	1.40	1.39	1.05	1.24	1.13
28	1.38	1.50	1.45	1.35	1.55	1.42
29	1.45	1.43	1.54	1.42	1.62	1.45
30	1.25	1.24	1.30	1.23	1.33	1.29
31	1.59	1.44	1.41	1.43	1.63	1.48

Table 8: Speedup relative to base case on AMD APU for LULESH

LULESH Layout ID	1 Thread	2 Threads	4 Threads
1	1.00	1.02	1.00
2	1.18	1.21	1.24
3	1.27	1.30	1.27
4	1.31	1.29	1.30
5	1.32	1.30	1.30
6	1.33	1.30	1.30
7	1.40	1.38	1.36
8	1.46	1.45	1.34
9	1.43	1.47	1.40
10	1.53	1.43	1.42
11	1.48	1.47	1.41
12	1.47	1.45	1.40
13	1.50	1.40	1.34
14	1.50	1.43	1.33
15	1.51	1.43	1.33
16	1.50	1.45	1.34
17	1.56	1.45	1.34
18	1.55	1.48	1.33
19	1.51	1.41	1.29
20	1.56	1.46	1.41
21	1.57	1.44	1.39
22	1.55	1.49	1.42
23	1.56	1.49	1.38
24	1.47	1.44	1.27
25	1.34	1.30	1.34
26	1.44	1.37	1.30
27	1.53	1.52	1.43
28	1.57	1.53	1.50
29	1.57	1.54	1.51
30	1.45	1.45	1.43
31	1.61	1.47	1.41

Table 9: Speedup relative to base case on Intel Sandybridge for LULESH

LULESH Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1	0.99	1.00	0.97	1.00	1.00
2	1.02	1.02	1.02	1.02	1.02
3	1.01	1.02	1.02	1.02	1.02
4	1.02	1.02	1.02	1.02	1.02
5	1.02	1.02	1.02	1.02	1.02
6	1.02	1.02	1.02	1.02	1.02
7	1.02	1.02	1.02	1.02	1.02
8	1.02	1.02	1.02	1.02	1.02
9	1.02	1.02	1.02	1.02	1.02
10	1.02	1.02	1.02	1.02	1.02
11	1.02	1.02	1.02	1.02	1.02
12	1.00	1.01	1.01	0.99	0.97
13	0.97	1.00	0.98	0.92	0.88
14	1.00	1.00	0.98	0.93	0.84
15	1.00	1.00	0.98	0.92	0.84
16	1.00	1.00	0.99	0.93	0.84
17	1.00	1.00	0.99	0.93	0.84
18	1.00	1.00	0.94	0.90	0.82
19	0.99	0.99	0.97	0.90	0.78
20	1.02	1.02	1.01	0.95	0.93
21	1.01	1.01	1.00	0.93	0.90
22	1.02	1.00	1.00	0.97	0.89
23	0.99	1.01	0.99	0.94	0.82
24	0.97	0.97	0.88	0.86	0.68
25	1.02	1.02	1.02	1.02	1.02
26	0.98	0.99	0.96	0.88	0.72
27	1.00	1.00	0.98	0.95	0.83
28	1.02	1.02	1.02	1.02	1.02
29	1.02	1.02	1.02	1.02	1.01
30	0.99	0.99	0.96	0.94	0.91
31	0.99	0.99	0.96	0.94	0.89

Table 10: Speedup relative to base case on BG/Q for LULESH

LULESH Layout ID	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
1	1.00	1.00	1.00	1.00	1.00	1.00	1.01
2	1.09	1.09	1.09	1.09	1.09	1.09	1.05
3	1.13	1.13	1.13	1.13	1.13	1.12	1.08
4	1.13	1.13	1.13	1.12	1.14	1.12	1.08
5	1.13	1.13	1.14	1.14	1.14	1.12	1.08
6	1.14	1.14	1.14	1.14	1.14	1.13	1.08
7	1.13	1.13	1.14	1.14	1.14	1.12	1.08
8	1.14	1.14	1.14	1.14	1.15	1.13	1.08
9	1.14	1.14	1.14	1.14	1.14	1.13	1.09
10	1.14	1.14	1.14	1.12	1.15	1.13	1.08
11	1.12	1.12	1.12	1.12	1.13	1.16	1.12
12	1.12	1.12	1.12	1.11	1.12	1.14	1.09
13	1.11	1.11	1.11	1.11	1.11	1.13	1.06
14	1.11	1.13	1.11	1.11	1.11	1.13	1.06
15	1.11	1.11	1.11	1.11	1.11	1.13	1.06
16	1.11	1.11	1.12	1.12	1.12	1.14	1.02
17	1.11	1.11	1.12	1.12	1.12	1.14	1.02
18	1.11	1.11	1.11	1.12	1.12	1.14	1.04
19	1.11	1.11	1.12	1.11	1.11	1.12	1.00
20	1.13	1.13	1.13	1.13	1.14	1.16	1.08
21	1.13	1.13	1.13	1.13	1.13	1.15	1.07
22	1.13	1.13	1.13	1.13	1.13	1.15	1.05
23	1.13	1.13	1.13	1.13	1.13	1.13	1.04
24	1.12	1.12	1.12	1.11	1.10	1.09	0.96
25	1.13	1.13	1.13	1.13	1.14	1.13	1.08
26	1.11	1.11	1.12	1.11	1.10	1.10	0.98
27	1.14	1.14	1.14	1.14	1.13	1.09	0.97
28	1.14	1.14	1.14	1.15	1.15	1.14	1.09
29	1.14	1.15	1.15	1.15	1.15	1.14	1.09
30	1.13	1.13	1.14	1.14	1.14	1.12	1.08
31	1.14	1.15	1.15	1.15	1.15	1.13	1.10

which the reference occurs. Multiple references to the same array in a loop are not distinguished; however, we do keep track of the types of accesses that occur: R, W, R/W. If a reference does not appear in a loop, we use its function name, to capture situations where the function call occurs inside a loop.

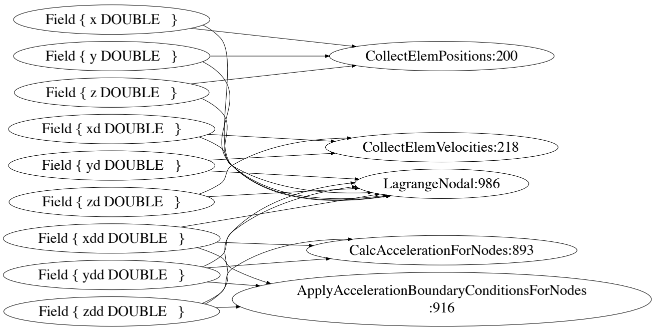


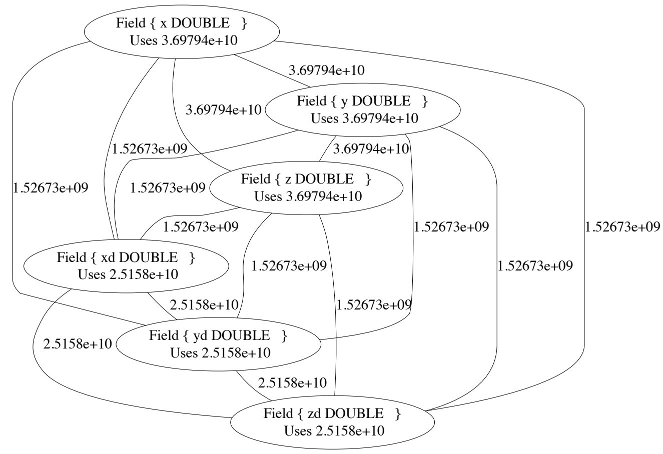
Figure 11: Sample Use Graph

Our Use graph is a bipartite graph *G=(U,V,E)* where U is set of arrays, V is set of uses of array references (function name and innermost loop) and E denotes a given array is used in a function of source program. Figure 11 shows a small subset of a sample Use graph. The left entries (U) denoted by Field corresponds to the arrays as specified in the user specification file. The array data type is also mentioned in the entry. The right entries (V) denote function names with loop statement number. This graph aids in easy identification of common array accesses across the loops. Use graph helps in processing other components of the automated layout algorithm to generate the best layout. For example, if two arrays never share a common use, they are not likely candidates for merging. From the sample graph, it is clear that merging of arrays for layouts is a non-trivial problem. For example, arrays x, y and z are exclusively used in CollectElemPositions function and jointly used in LangrangeNodal function with arrays xd, yd, zd, xdd, ydd and zdd. Merging these two arrays sets would lead to better locality for LangrangeNodal, but lead to additional lines being fetched in CollectElemPositions. Complexity of automated layout also increases with more arrays and function references. These constraints pose a difficult choice to the automated algorithm. An expert programmer will have to make such choice to select an appropriate layout. Even if the programmer wishes to experiment with a manual layout, he may wish to generate a Use graph to better understand the relationship amongst arrays in an automated way.

6.2 Affinity Graph

The Use graph shows the relationship between arrays and its function uses. However, in order to quantify the merging between arrays, we need a relationship amongst arrays. An Affinity graph represents the commonality of uses between different arrays across all functions. The number of common iteration accesses determines closeness across the array group. Our Affinity graph builds upon the Use graph to determine the common function accesses.

Figure 12 shows a sample affinity graph. Each node represents a unique array with the number of uses across the whole program. Link value between the arrays denotes the number of common uses between them. Profiling the relevant loops across the functions collects the number of uses. With this information and Use graph, we can construct the affinity graph with all the populated values. In the sample graph, array x and y, both have number of uses as 3.69e+10 and share a link with value 3.69e+10. This indicates that both arrays have all the references appearing together and are strong candidates for array merging. Looking at array z and xd, we find that they have a lot of common accesses but do have other uses as well. Based on constraints, we may or may not want to merge these arrays together.



Field { x DOUBLE } Uses 3.69794e+10
Field { y DOUBLE } Uses 3.69794e+10
Field { z DOUBLE } Uses 3.69794e+10
Field { xd DOUBLE } Uses 2.5158e+10
Field { yd DOUBLE } Uses 2.5158e+10
Field { zd DOUBLE } Uses 2.5158e+10
3.69794e+10
3.69794e+10
3.69794e+10
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
1.52673e+09
2.5158e+10
2.5158e+10
2.5158e+10

Figure 12: Sample Affinity Graph

6.3 Affinity Index

The Affinity Graph helps us understand the relationship between the arrays and their uses. However, we need a mathematical term to denote the affinity i.e. closeness between the arrays. Affinity Index is defined as the closeness factor among a set of arrays in a subgraph. Our algorithm uses the affinity index by merging two array groups (or a subgraph), if the affinity index of a subgraph meets a threshold.

Before defining affinity index, let us define another term link index. Given two array references x and y with its uses, Use_x and Use_y and a link between them with a value LV, then,

$$LinkIndex = \frac{LV}{Use_x} + \frac{LV}{Use_y} \tag{1}$$

In Figure 12, $LinkIndex = 2$ for arrays x and y, whereas $LinkIndex = 0.10$ for arrays xd and z. We state a Link Value property as follows:

Property 1: $LinkValue(LV) \leq min(Use_x, Use_y)$

Now, we define Affinity Index in terms of Link Index from Equation 1 as follows,

$$AffinityIndex = \frac{\sum_{i=1}^{numLinks} LinkIndex_i}{2 * numLinks} \tag{2}$$

where numLinks denotes the number of links present in a subgraph. In Figure 12, AffinityIndex=1 for arrays x and y. Note that AffinityIndex still remains 1 for arrays x, y and z as well, since all of them always share a common loop.

6.4 Cache-Use Factor(CUF)

Affinity index indicated association between arrays. To indicate the cache impact of a possible merging of array groups, we define a second term Cache-Use Factor. We explicitly use dynamic loop counts from the profiled data with Use graph to calculate cache-use factor. This factor indicates the appropriate uses of cache lines fetched during a loop execution. On merging two arrays groups, cache-use factor may be lowered since all array accesses might not be used across the loops. For example in Figure 11, merging of array groups x,y,z and xdd,ydd,zdd might lead to a low cache-use factor, since these groups just share a single loop LagrangeNodal in common.

Before defining cache-use factor, we define another term cache-loop factor (CLF). Given an array group A (i.e. set of arrays) and loop L, cache-loop factor is defined as follows,

$$CLF = \frac{|\{A\} \cap \{\ \textit{array references in L}\ \}|}{LC} \tag{3}$$

where LC denotes the loop count of loop L. Now, we define Cache-Use Factor in terms of Cache-Loop Factor as follows,

$$CUF = \frac{\sum_{i=1}^{numLoops} CLF_i}{\sum_{i=1}^{numLoops} LC_i}, \textit{where } LC_i = 0 \textit{ if } CLF_i = 0 \tag{4}$$

For the base case, where all array are separate, we have $CUF = 1$. However, as we merge array groups the CUF value might lie somewhere between 0 and 1, with 1 indicating better cache line use. This might lead us to refrain from merging some arrays, as we desire the highest CUF. However, merging helps in better register use and locality of elements as we have seen in Section 5.

6.5 Automatic Data Layout Algorithm

Our Automated Data Layout Algorithm uses the affinity index, cache-use factors, and platform characteristics to produce a meta file that contains the recommended data layout. Algorithm 1 shows our Automated Data Layout Algorithm. To begin, each array in the Field Specification is its own ArrayGroup. The algorithm compares all pairs of ArrayGroups to determine the viability and value of merging each pair. The pair with the highest combined value (affinity and cache-use factor)are merged into one new group. The two selected array groups are merged into one new group. This process is repeated until the best candidate pair for merging falls below the acceptable merge threshold. After the final grouping is determined, each group's arrays are sorted based on data type, to better pack them. The final step performs cache line splitting to efficiently utilize each cache line fetch for the target platform.

The evaluation of the viability and value of merging two candidate ArrayGroups considers two factors. The first consideration examines reads versus writes to an ArrayGroup. Our manual results (Section 5) showed that grouping arrays written to frequently with arrays that are read only decreases performance significantly. Our current heuristic prohibits combining any ArrayGroup when that group has more than 2× more writes than reads. The second consideration for merging ArrayGroups computes a new Affinity index and cache use factor for the proposed combination. If both values are greater than our established thresholds, the ArrayGroups are viable for merging. From our empirical results, we have chosen, an *affinity threshold* = 0.62 and a *Cache Use threshold* = 0.52 for our algorithm. A detailed analysis to study the effects of varying these parameters across architectures and benchmarks must be left as future work.

Platform	IRSmk		LULESH	
	Best Manual Layout	Automated Layout	Best Manual Layout	Automated Layout
Power7-8Threads	4.70	4.67	1.43	1.58
Power7-32Threads	5.02	4.93	1.47	1.19
AMD APU-4Threads	1.46	1.43	1.50	1.46
Sandybridge-8Threads	1.11	1.10	1.02	0.96
Sandybridge-16Threads	1.05	1.03	1.02	0.91
BG/Q-64Threads	2.20	2.08	1.10	1.04

Table 11: Best Manual Layout vs. Automated Layout speedup as compared to base layout

7. AUTOMATIC DATA LAYOUT RESULTS

Table 11 shows the results for best manual layouts and automated layouts in comparison to base layout. Results demonstrate that automated layouts were within 95-99% of best manual layout for IRSmk, and within 89% of best manual layout for LULESH except on the Power 7 where it performs better than the manual layouts on 8 threads and performs about 20% worse on 32 threads. These results prove the effectiveness of our automated results. Our automated results also show more significant improvements for on-chip accesses across the architectures (8 Threads on Power7 and Sandybridge) in comparison to off-chip counterparts. In one particular case, 8 Threads on Power7 for LULESH, automated layout improved performance as compared to manual layouts. These architectures exhibit NUMA behavior, which our automated algorithm doesn't consider for this work. We believe that either extending our algorithm to incorporate memory allocation or using NUMA libraries for memory allocation would further increase layout performance on these architectures. However, for this work, we only used the default memory allocation provided on these systems.

8. RELATED WORK

Past research has proposed various data layout optimization techniques [7–9, 11]. Here, we present a brief survey of past work, focusing on aspects that are most closely related to our work.

Zhang et al. [24] introduced a data layout framework that targets on-chip cache locality, specifically reducing shared

Algorithm 1 Automated Data Layout Algorithm

```
procedure IsMerge($ArrayGroup_i$, $ArrayGroup_j$)
    if SignificantWrites($ArrayGroup_i$) || SignificantWrites($ArrayGroup_j$) then
        return $false$
    end if

    if AffinityIndex ($ArrayGroup_i$, $ArrayGroup_j$) $\geq$ AFFINITYTHRESHOLD &&
      CacheUseFactor($ArrayGroup_i$, $ArrayGroup_j$) $\geq$ CACHEUSETHRESHOLD) then
        return $true$
    end if

    return $false$
end procedure

procedure AutoDataLayout(ArrayGroupList)

    while $change = true$ do
        $change \leftarrow false$

        for $i \in ArrayGroupList$ do
            for $j \in ArrayGroupList$ do
                if IsMerge($ArrayGroup_i$,$ArrayGroup_j$) then
                    $change \leftarrow true$
                    $affinityIndex(i, j) \leftarrow$ AffinityIndex($ArrayGroup_i$, $ArrayGroup_j$)
                    $cacheUseFactor(i, j) \leftarrow$ CacheUseFactor($ArrayGroup_i$, $ArrayGroup_j$)
                end if
            end for
        end for

        if $change = true$ then
            $index \leftarrow$ getBestAffinityCacheUsePair($affinityIndex, cacheUseFactor$)
            $ArrayGroup_{ij} \leftarrow$ mergeGroups($ArrayGroup_{\text{index-i}}$, $ArrayGroup_{\text{index-j}}$)
            $ArrayGroupList \leftarrow ((ArrayGroupList - ArrayGroup_{\text{index-i}}) - ArrayGroup_{\text{index-j}}) \cup ArrayGroup_{ij}$
        end if
    end while

    sortGroups(ArrayGroupList)
    splitCacheLine(ArrayGroupList)
    return ArrayGroupList
end procedure
```

cache conflicts while observing data patterns across threads. Using polyhedral analysis, their framework rearranges data layout tiles to reduce on-chip shared cache conflicts. However, their optimization currently works with single arrays. In contrast, our approach works on merging multiple arrays and operates at the element level rather than tiles.

Henretty et al. [13] presented a data layout framework to optimize stencil operations on short-SIMD architectures. Their work specifically targets stream alignment conflicts on vector registers and uses a dimension transposition method (nonlinear data layout optimization) to mitigate the conflicts. In comparison, our approach works for more general applications, not just stencil code. Also, our work did not specifically address the impact of data layout on vectorization.

Ding and Kennedy [11] introduced a data-regrouping algorithm, which has similarities to our work on automatic selection of data layouts. Their compiler analysis merges multidimensional arrays based on a profitability cache analysis. Dynamic regrouping was also provided for layout optimization at runtime. Experimental results show significant improvement in cache and TLB hierarchy. However, their results were all obtained on uniprocessor systems and it is unclear how their approach works in the presence of data aliasing.

Raman et al. [21] used data layout transformations to reduce false sharing and improve spatial locality in multithreaded applications. They use an affinity based graph approach (similar to our approach) to select candidates. Interprocedural aliasing issues arising due to pointers is not addressed in this work. Our work is intended to explore data layout transformations more broadly, not just for false sharing and spatial locality. Using polyhedral layout optimization, Lu et al. [18] developed a data layout optimization for future NUCA CMP architectures. Their work reduces shared cache conflict on such architectures. Simulation results show significant reductions in remote accesses. Finally, a number of papers, [12, 15, 20, 23] have explored the integration of loop and data layout transformations.

To the best of our knowledge, our work is the first to support both user-specified and automatic AoS and SoA data layout transformations, while allowing the user to provide a data layout specification file. Our results on the LULESH mini-application demonstrates the importance of data layout transformations on modern multicore processors.

9. CONCLUSIONS

This paper establishes the foundation for a new approach to supporting portable performance of scientific codes across HPC platforms. The upgraded TALC source-to-source transformation tool permits application developers to maintain one "neutral" data layout source code and explore architecture specific array layouts. The new automated portion of TALC can analyze the original source code based on platform characteristics and produces a new source code with new array data layouts ready to be compiled and run on that system. The results for the two test codes show that maunal layouts improve performance by 1.10× to 22.23× for IRSmk and 1.02× to 1.82× for LULESH with results varying with thread count and architecture. The automated algorithm resulted in performance of 95-99% of the best layout manual layout for IRSmk. For LULESH the automated approach was better than any manually tried layout for 8 threads on the Power 7 and within 10% of the best layout on all other processors except for 32 threads of the Power 7 where NUMA effects might cause the automated layout selection to be sub-optimal.

This work represents a first step in this approach to portable performance. Opportunities for future work abound. One direction expands the flexibility of constraints on the original source code to include manipulation of multi-dimensional arrays. For many applications, it could be highly useful to reorder dimensions of multi-dimensional arrays, sub-divide those arrays, and interleave them with compatible arrays of different dimensions. Another direction of future work is to study a broader range of platforms, particularly GPU systems, where there can be mirrored copies of arrays that might benefit from different layouts on the heterogeneous cores. We also need to study a much richer range of scientific source codes. The conditions set up by different codes will almost certainly require improvements to the heuristics current in the automated TALC section. A key aspect of looking at different codes will be the examination of actual array subscript usage within loops to compute more accurate affinities. We look forward to pursuing these challenges.

10. ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

11. REFERENCES

[1] AMD64 Architecture Programmer's Manual Volume 2: System Programming.

[2] ASC Sequoia Benchmark Codes. https://asc.llnl.gov/sequoia/benchmarks/.

[3] ROSE Compiler Infrastructure. http://rosecompiler.org/.

[4] TALC Infrastructure. https://wci.llnl.gov/codes/talc/.

[5] Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, Livermore, CA, USA, July 2011. https://computation.llnl.gov/casc/ShockHydro.

[6] AMD. Software Optimization Guide for AMD Family 15h Processors. Technical Report 47414, January 2012.

[7] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pages 139–149, New York, NY, USA, 1998. ACM.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[9] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 252–262, New York, NY, USA, 2006. ACM.

[10] I. Chung, C. Kim, H.-F. Wen, G. Cong, et al. Application data prefetching on the IBM blue gene/Q supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012*

International Conference for, pages 1–8. IEEE, 2012.

[11] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 149–163, London, UK, UK, 2000. Springer-Verlag.

[12] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 38–, Washington, DC, USA, 2001. IEEE Computer Society.

[13] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. Technical Report 248966-026, April 2012.

[15] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 95–, Washington, DC, USA, 1999. IEEE Computer Society.

[16] I. Karlin, J. McGraw, J. Keasler, and C. Still. Tuning the LULESH Mini-app for Current and Future Hardware. In *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, December 2012.

[17] J. Keasler, T. Jones, and D. Quinlan. TALC: A Simple C Language Extension For Improved Performance and Code Maintainability. In *9th LCI International Conference on High-Performance Clustered Computing*, April 2008.

[18] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 348–357, Washington, DC, USA, 2009. IEEE Computer Society.

[19] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[20] M. F. P. O'Boyle and P. M. W. Knijnenburg. Efficient parallelization using combined loop and data transformations. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 283–, Washington, DC, USA, 1999. IEEE Computer Society.

[21] E. Raman, R. Hundt, and S. Mannarswamy. Structure layout optimization for multithreaded programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 271–282, Washington, DC, USA, 2007. IEEE Computer Society.

[22] B. Sinharoy, R. Kalla, W. Starke, H. Le, R. Cargnoni, J. Van Norstrand, B. Ronchetti, J. Stuecheli, J. Leenstra, G. Guthrie, et al. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[23] M. Taylan Kandemir. Improving whole-program locality using intra-procedural and inter-procedural transformations. *J. Parallel Distrib. Comput.*, 65(5):564–582, May 2005.

[24] Y. Zhang, W. Ding, J. Liu, and M. Kandemir. Optimizing data layouts for parallel computation on multicores. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 143–154, Washington, DC, USA, 2011. IEEE Computer Society.